ERSP Project Proposal: Querying Graphs and their Representations

Wyatt Hamabe, Will Corcoran, Niyati Mummidivarapu
Advisors: Dr. Ambuj Singh, Danish Ebadulla

December 2023

1 Research Context and Problem Statement

Over the last decade, graphs have become an increasingly common way to store data. With that came the development of graph databases (GraphDBs) and graph query languages. These databases and languages are the graph analog to common SQL. Concurrently, machine learning and neural networks expanded to accept graph-based input, leading to Graph Neural Networks (GNNs). The task of a GNN, like many machine learning models, is to learn information from the data. A few examples are recommendations from connections in a graph (technically known as link prediction), node classification, or learning information about the graph structure (often used in chemistry and related fields).

1.1 Background and Motivation

Similar to other neural networks, GNNs receive data as input. These data points are transformed into a representation space (also known as an embedding or embedding space), which attempts to capture a meaningful relationship between points. To analyze these representations, one must use a technique called vector similarity search. Given a point of interest, vector similarity search identifies the k most similar data points (in this case, nodes). This is known as a k-nearest neighbors search (kNN).

Graph neural networks capture relationships between nodes and edges beyond what would be recognizable from a topological view of a graph. [13] For context, graphs are often referred to as a topological or structural space. The process of analyzing spaces, either topological or embedding, is called a view. The analysis of the difference between topological and embedding spaces can help uncover useful insights about the graph. Typically, this is done manually, resulting in a workflow that is tedious and inefficient.

Database management systems and their respective query languages help filter and analyze large amounts of data. Just like SQL for those platforms, graph database management systems (GraphDBs) and their query languages help interpret relationships in the topological view of a graph. However, these softwares do not support analysis of the representation space alongside a graph.

Frameworks used for vector similarity search, like Meta's FAISS, are designed to be scalable and efficient with the capability of analyzing vector similarity for extremely large datasets. Unfortunately, while having the ability to analyze representation space, these softwares lack the services that a graph database provides.

We propose a framework that marries the analysis capabilities of a graph database along with the power of k-nearest neighbors search from a vector similarity search technology. Our implementation will allow the user to understand the relationships truly captured within graphs. This can range from a high-level examination of the viability of different GNN architectures in capturing these relationships, or a low-level identification of nodes and edges that influence the representation or clustering of nodes. As a final product, we will develop an SQL-analogous query language, built upon existing technologies, that can query from

both the topological and representation view of a graph.

This framework has a wide range of applications. Some domains where this might be useful are:

- **Node Vulnerability Analysis**: Using our work to identify fraudulent nodes and possible adversarial attacks can help detect malicious agents quickly. kNN search along with a graph query can highlight outlying or irregular nodes. This improves the security of GNNs, graphs, and their data.
- **Feature Analysis**: There are scenarios where it can be helpful to see which features have a larger influence in outcome or predictions. This can be achieved using a query upon the GraphDB for specific features and pairing this will the respective kNN clusters. Results will accentuate the most impactful features.
- **Graph Structure and Learning Outcomes:** Certain graph patterns and structures can impact the outcome of a GNN. Specific patterns can be queried from the GraphDB and the appropriate kNN clusters can be analyzed.
- **GNN Model Comparisons**: Different GNN architectures can be analyzed based on their kNN information for the same graph through a query. This has benefits in larger systems with many nodes and edges.

1.2 Literature Review

Often, GNNs are built with multiple layers. Each layer has a slightly different formula, or modification, affecting the input in specific ways. A GNN process consists of three steps: *aggregation, combination,* and *classification.* Message passing is a common form of aggregation. Typically, data is "passed as a message" to its surrounding nodes. After enough passes, similar nodes begin to have homogeneous data. Combination has a similar functionality, however, it tends to be exclusively between two nodes, whereas aggregation can be between any amount of nodes. Classification functions are normally Boolean functions to evaluate the correctness of the GNN. Given some values for correctness, the GNN performs iterations, continually improving the model. [8]

Currently, a common issue with many graph neural networks is that the internal embeddings are not easily interpretable. To address this issue an approach named CorGIE (Corresponding a Graph to Its Embedding) [10] has been proposed. It aims to provide a clearer understanding of how a GNN processes a graph.

CorGIE is an interactive multi-view interface tool that visualizes these embeddings. It abstracts the data and tasks, meaning it simplifies and generalizes them to make the process more understandable. The k-hop graph layout is a key feature of CorGIE. This layout shows nodes that are 'k' links away from a given node (known as topological neighbors) and their clustering structure. This helps us understand how nodes are grouped or connected within the graph itself. Along with this, CorGIE provides a graphic depiction of the embedding space (referred to as latent space by CorGIE). These are two different views, giving a before-and-after look into the representations.

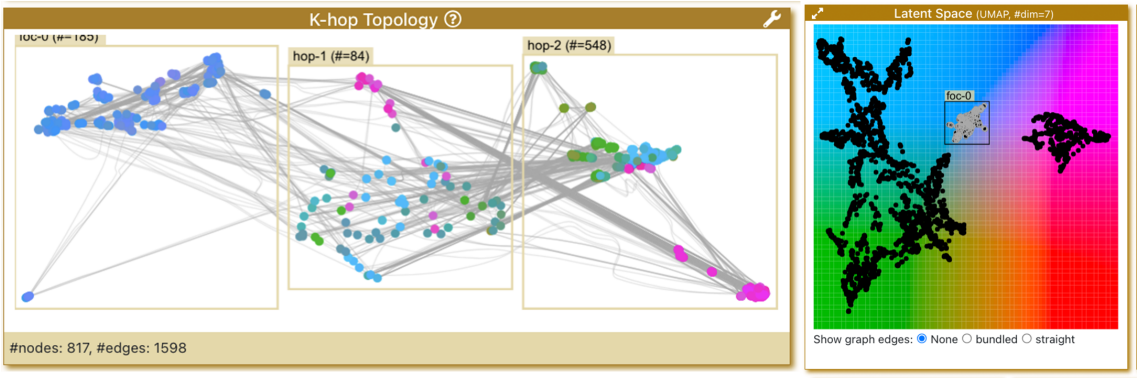

Figure 1: *[Left] On the Cora dataset, CorGIE shows the "k-hop" topology visualizing the graph structure and the nodes in 2-hop range of the selected cluster (darker blue) of nodes. [Right] CorGIE shows the embeddings of a GNN, with the Cora dataset as input, in latent space.*

While CorGIE is a good visualization tool, it fails in many important ways. To begin, the maximum size of a graph allowed by CorGIE is 20,000 nodes. Many graphs are much larger than that, which renders CorGIE useless. CorGIE doesn't support a querying functionality. Instead, you are limited to different visualizations, none of which have any applicable output for an advanced program. Along with this, there is no k-nearest neighbors tool. The *latent space* module does a decent job of visualizing kNNs, however, it fails to have any level of customizability and advanced search. Also, there is no correlation with a GraphDB, leaving CorGIE impractical. Rather, the user must upload their data in JSON format, which is unrealistic. CorGIE hasn't been maintained in multiple years, so much of the functionality is outdated and lacks support.

For further discussion on similar topics: [8] explains how we can query a class of embeddings by passing different inputs and gauging the resulting vectors. The authors introduce GEL (Graph Embedding Language), which lays out syntax-based examples for ways to query on resultant embedding matrices. While we are focused on querying from the original graph itself, it still provides a quality foundation for syntax. Lastly, [5] explains how GNNs can better classify nodes in a graph by using First-Order Logic–a formal system used in mathematics to express statements about objects and their relationships. As we are not focused on GNN algorithms, but rather the information presented after these algorithms, this solution is not reasonable.

2 Proposed Solution

As a solution, we'll leverage three technologies, all of which were briefly described: graph databases and query languages, specifically ArangoDB and AQL to extract information directly from the graph; additionally, we'll employ FAISS, a vector similarity search tool by Meta, which gathers k-nearest neighbors. These are ubiquitous technologies, however, their intersection is our solution to the problem. Next, these technologies and their intended utilization are described in depth. Finally, we posit our anticipated solution, Graph-Learning Query Language, with the necessary functionalities.

2.1 Graph Databases and Query Languages

```
FOR x IN UNION_DISTINCT(

    (FOR y IN ANY 'movies/TheMatrix' actsIn
    OPTIONS { order: 'bfs', uniqueVertices: 'global' }
    RETURN y._id),

    (FOR y IN ANY 'movies/TheDevilsAdvocate' actsIn
    OPTIONS { order: 'bfs', uniqueVertices: 'global' }
    RETURN y._id)

) RETURN x
```

Figure 2: *Example of an AQL (ArangoDB Query Language) query that can locate actors who were in "The Matrix" or "The Devils Advocate." [1]*

Graph databases have exploded in popularity over the past few decades. Neo4j and ArangoDB have emerged as the two front runners. Each represents two distinct approaches to graph-structure data.

Neo4j, exclusively a graph database, is renowned for its graph processing capabilities, which make it highly efficient for exploring intricate networks of data relationships. Cypher, the query language of Neo4j, focuses on syntax specific to graph patterns. This specialization makes Neo4j a leader in the GraphDB ecosystem. [11]

ArangoDB, conversely, uses a multi-model approach, which amalgamates graph data with documents and key-value data models. It fails to provide as many graph-specific functions as Neo4j. However, it replaces these with versatility–allowing users to employ graph, document, and key-value commands within a single query. Nonetheless, ArangoDB has grown to focus on its graph prowess, providing many complex traversals and algorithms. [4]

ArangoDB significantly outperforms Neo4j in two very important categories: scalability and efficiency. Surveys of popular GraphDBs and their efficiency indicate that ArangoDB dominates Neo4j in *all* observed categories. [3] Along with that, ArangoDB implements proprietary solutions to scalability challenges using *SmartGraphs*. Neo4j fails to provide a solution. [2]

Many graph neural network problems require large storage space. Often, graphs can grow to billions of nodes. Our solution prioritizes support for such graphs, hence, we intend to use ArangoDB (and AQL) as our counterpart. Not to mention, ArangoDB is shown to have superior efficiency. However, it is worth noting that all functionality we implement or describe can be done with Neo4j, just not with the same scalability or efficiency.

2.2 Vector Similarity Search

```
# we want to see 4 nearest neighbors
k = 4
D, I = index.search(xb[:5], k)
print(I)
print(D)
```

Figure 3: *Example FAISS query to find the four nearest neighbors of an arbitrary vector, xb [7]*

Vector similarity search (VSS), specifically k-nearest neighbors search, is a powerful tool. Generally speaking, VSS calculates the distance between vectors, under the assumption that similar vectors are placed in the same cluster (or area). Vector similarity search has been implemented in various Python packages, one of which is Meta's Facebook AI Similarity Search (FAISS), which we will use as our foundation.

While traveling through a graph neural network, the input matrix (data) is changed at each layer. Typically, this happens through an aggregation process, similar to the one described in Section 1.2. To find the similarity between two nodes (represented as vectors) on the graph, a calculation is performed. Often, this is the task of the GNN. A task is the final "goal" of the designed GNN. We'd like to be able to do this similarity search on the embeddings produced by this GNN (as a result of some task). We can do this using FAISS' kNN search and supplying a set of vectors (by decomposing some matrix, M_d). FAISS defines this calculation as:

"Given a set of vectors, $\{x_1, ..., x_n\}$, [and a vector x, find d_i that is the minimum distance between x and some vector x_i],

$$d_i = argmin_i ||x - x_i||$$

where $||\cdot||$ is the the Euclidean distance (L2)." [9]

FAISS uses this calculation to determine the k-nearest neighbors for a vector, x. Given all values of d_i, finding the k-nearest neighbors is quite trivial. FAISS' major advancement was the use of GPU acceleration to speed up the querying time, as these are incredibly expensive calculations, especially as dimensions can get into the billions. Nonetheless, they claim to be 8.5x [12] times faster than the next quickest vector similarity search tool, making them the leading tool for us to use.

Importantly, FAISS implements different methods to ascertain the k-nearest neighbors. Some of these provide 100% accuracy (e.g. *IndexFlatL2, IndexFlatIP*) by checking all vectors. FAISS calls this an "exhaustive search." Others sacrifice guaranteed accuracy for faster runtimes (e.g. *IndexHNSWFlat, IndexIVFFlat*), by considering only a subset of vectors. Consequently, there is a correctness-efficiency tradeoff, which must be considered in the proposed solution.

2.3 Graph-Learning Query Language

We propose Graph-Learning Query Language, GLQL, to query across two views. The first is a topological perspective using AQL, and the second is a representation view using FAISS. As stated in Sections 2.1 and 2.2, both of these operations can be completed with pre-existing frameworks. We plan to write a query language that formulates string-based queries across both views (topological and embedding). Per standard operations of a query, the respective data shall be returned in a relatively quick matter, in a form that can be interpreted and transmitted with ease.

```
FOR researcher IN 'academics'
    FILTER researcher.department == 'computer science'
    FOR collaborator IN 1 .. 3 OUTBOUND researcher
            GRAPH 'collaborationGraph'
    FILTER SIMILAR_TO(collaborator, 'researchInterests',
            researcher, 10)
    RETURN { researchName: researcher.name,
            collaboratorName: collaborator.name }
```

Figure 4: *An example of a proposed query that finds potential research collaborators within 3-hops. Then, it verifies that these collaborators are with the 10-most similar researchers.*

The code block in *Figure 4* shows, for each node, $researcher$, in a graph, $academics$, check to verify that they are in the computer science department. If they are, get each $collaborator$ within 3-hops of $researcher$. Check to see if this $collaborator$ is in the 10-nearest neighbors to $researcher$ based on research interests. If this is the case, return them as a pair.

At the moment, there are two candidate solutions. We plan to implement both and evaluate their correctness and efficiency.

1. Query on the initial graph, limit the GNN to only the resulting vectors, using FAISS to find the k-nearest neighbors, return.
2. Query on the initial graph, query on the GNN for all k-nearest neighbors, intersect this data, return.

As mentioned, we will develop a query language. This can best be solved by creating a strict-syntax language, which can be passed as a string to, preferably, a Python function. Along with the string, allowing users to specify a specific FAISS implementation gives them sovereignty over accuracy and runtime.

This Python function will incorporate an API, sending the string (and optionally, implementation specifics) to the respective C++ code, to handle the query. FAISS is faster implemented within C++, so for the benefit of running the software efficiently, using a C++ backend serves tremendous benefits. Generalization to other languages, databases, and scenarios is of lesser importance due to the ease of the implementation after the initial.

3 Evaluation and Implementation Plan

In the future, portions of the project may pivot, so it's important to outline a procedure to stay on track and evaluate the completed work. Here, we devise an evaluation method based on accuracy, runtime, functionality, and scalability, along with a 2-quarter plan.

3.1 Evaluation

3.1.1 Correctness of Queries

Queries should be 100% accurate, however, as mentioned in the FAISS description (Section 2.2), there are tradeoffs between accuracy and efficiency. FAISS has a brute force algorithm, *IndexFlatL2*, which according to Meta, is "exhaustive," indicating correctness. However, *IndexFlatL2* is much less efficient than *IndexHNSWFlat*, which is "not exhaustive," sacrificing some accuracy for efficiency. ArangoDB uses proven-correct graph algorithms to query data (depth-first search, breadth-first search, etc.). These algorithms are known to be correct, returning 100% accurate results. Therefore, depending on the FAISS algorithm type, accuracy and correctness can vary.

We will do statistical analysis to find the general accuracy of different FAISS algorithms. These algorithms consequently affect the correctness of GLQL. Hence, treating *IndexFlatL2* as the baseline (guaranteed correctness) shall be sufficient. We will compare this baseline against *IndexHNSWFlat*, *IndexIVFFlat*, *IndexLSH*, *IndexIVFScalarQuantizer*, and other common methods, by calculating the median error with regards to *IndexFlatL2*.

The success of queries can be classified by those algorithms that return correct results within some error, e_{median}. This can be relaxed or tightened depending on the intentions (strict or loose recommendations using kNN). Different levels of accuracy will be accepted depending on use cases, hence, a single accuracy metric isn't realistic.

3.1.2 Runtime

Use case heavily affects choices to prioritize efficiency or accuracy. For inspiration, FAISS uses the metric of $ms/vector$. [12] This is considered a viable measurement of efficiency based on the dimensionality of the matrix and its associated graph neural network. Exact runtime goals are uncertain as no testing has been done. However, the retrieval time should be reasonable for usage in social networking and similar fields. As is the case with FAISS, we presume that faster retrieval times will result in decreased accuracy, likewise, slower retrieval times will result in increased accuracy.

3.1.3 Functionality

While there is no official metric, the breadth of queries that can be handled is crucial. If the language only works under certain circumstances, it lacks usability. We intend to design a language that has the capability of being a ubiquitous tool for GNN developers worldwide. This requires adapting to the maximum amount of queries possible.

3.1.4 Scalability

Similar to functionality, if the language only works on graphs of a certain size, it fails in practicality. This is one of CorGIE's main failures. To test this, we plan on using small graphs (`TexasDataset` and `CornellDataset`), medium graphs (`CoraGraphDataset` and `ActorDataset`), and large graphs (`YelpDataset` and `AMDataset`) from DGL, a popular graph dataset package. [6] We will record any bugs, bottlenecks, or inefficiencies encountered, and then fix them in order to ensure scalability.

3.2 Implementation Plan

Fall 2023:

- Weeks 7-10:
  - Submit a high-quality rough draft by the start of Week 8.
  - Familiarize and practice using the following tools:
    1. PyTorch (with focus on GNNs)
    2. CorGIE (representing latent space of GNNs)
    3. FAISS (focusing on how it calculates k-nearest neighbors)
    4. ArangoDB (understanding how the query language works and how we can apply similar techniques)
  - Submit the *final* proposal on Wednesday, December 6^{th}.
  - Create, orchestrate, and plan slide-deck for the culminating presentation during finals week.

Winter 2024:

- Weeks 1-4:
  - Build a program, given an ArangoDB table along with its respective GNN embedding and a basic query, which does the following:
    1. Run the query relative to the ArangoDB table
    2. Run the k-nearest neighbor query on the GNN embedding
    3. Return the intersection of these two queries.
- Weeks 5-10:
  - Syntactically design a query language with the following traits:
    1. Similar enough to AQL to transfer nicely throughout queries.
    2. Ease-of-use k-nearest neighbors querying.
  - Implement the language:
    1. Build APIs to forward the data between databases and programming languages.
    2. Structure queries so they can be done easily inside of a language, rather than having to write in terminal.
  - Rename query language. GLQL is a place holder.
  - Complete reproducibility documents so other researchers can use the technology to test.

Spring 2024:

- Weeks 1-3:
  - Expand the project to include other avenues, which were outlined by Professor Singh earlier in the quarter, these include:
    1. Finding subgraphs and node outliers in GNN embeddings
    2. Compare across different GNN models, looking for patterns in the embeddings
    3. Consider different ways to reduce distortions in embeddings
- Weeks 4-7:
  - If writing a paper for conference, as mentioned by grad. mentor as a possibility:
    1. Go through a very similar process as this paper.
    2. Augment the paper for the specific conference / journal.
  - If not, continue with additional research from Weeks 4-7 of Winter Quarter.
- Weeks 8-10:
  - Begin to wrap up everything over the previous two quarters.
  - Design the poster and presentation for the final assembly in June.

References

[1] ArangoDB. AQL Example Queries on an Actors and Movies Dataset. https://docs.arangodb.com/3.11/aql/examples-and-query-patterns/actors-and-movies-dataset-queries/, 2023. [Online; accessed 14-November-2023].

[2] ArangoDB. Comparing ArangoDB AQL to Neo4j Cypher. https://arangodb.com/learn/graphs/comparing-arangodb-aql-neo4j-cypher/, 2023. [Online; accessed 2023].

[3] ArangoDB. NoSQL Performance Benchmark 2018 – MongoDB, PostgreSQL, OrientDB, Neo4j and ArangoDB. https://arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/, 2023. [Online; accessed 2023].

[4] ArangoDB. Vision. https://arangodb.com/about-us/: :text=ArangoDB [Online; accessed 2023].

[5] P. Barceló, E. V. Kostylev, M. Monet, J. Pérez, J. L. Reutter, and J.-P. Silva. The expressive power of graph neural networks as a query language. *SIGMOD Rec.*, 49(2):6–17, dec 2020.

[6] DGL. dgl.data. https://docs.dgl.ai/api/python/dgl.data.html/, 2023. [Online; accessed 2023].

[7] M. Douze. Getting some data. https://github.com/facebookresearch/faiss/wiki/Getting-started/, 2020. [Online; posted 28-June-2020].

[8] F. Geerts. A query language perspective on graph learning. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '23, page 373–379, New York, NY, USA, 2023. Association for Computing Machinery.

[9] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[10] Z. Liu, Y. Wang, J. Bernard, and T. Munzner. Visualizing graph neural networks with corgie: Corresponding a graph to its embedding. *IEEE Transactions on Visualization and Computer Graphics*, 28(6):2500–2516, 2022.

[11] W. L. Lju Lazarevic. Overview of the Neo4j Graph Data Platform. https://neo4j.com/developer-blog/overview-of-the-neo4j-graph-data-platform/, 2021. [Online; accessed 2023].

[12] Meta. Faiss: A library for efficient similarity search. https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/, 2017. [Online; posted 29-March-2017].

[13] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications, 2021.